Apuntes U 4

Tema ***Algoritmos de Ordenamiento y Número Aleatorio***

"Estar preparado es importante, saber esperar lo es aún más, pero aprovechar el momento adecuado es la clave de la vida."
Arthur Schnitzler

¿Qué es el ordenamiento?

Para conseguir mayor eficiencia en el tratamiento de la información, tanto en el ámbito de almacenamiento, la información debe tener algún tipo de ordenamiento.

> En computación *y matemáticas* un algoritmo de ordenamiento es un algoritmo que pone elementos de una lista o un vector en una secuencia dada por una relación de orden, es decir, ***el resultado de salida ha de ser una permutación,*** *(o reordenamiento)* de la entrada que satisfaga la relación de orden dada.

Criterios:
- Numéricos: Ascendente/Descendente
- Lexicográficos: Ascendente/Descendente *(refiriéndonos a caracteres)*

Métodos de ordenamiento según su lugar de procesamiento:

Internos: Se denomina así porque se realiza directamente y completamente en memoria principal *(todos los objetos que se ordenan caben en la memoria principal de la computadora)* con lo que el proceso es más rápido.

Externos: Cuando no cabe toda la información en memoria principal y es necesario ocupar memoria secundaria *(disco)*. El ordenamiento ocurre transfiriendo bloques de información a memoria principal en donde se ordena el bloque y este es regresado, ya ordenado, a memoria secundaria. En consecuencia es mas lento.

Métodos de ordenamiento según su forma de procesamiento:

Iterativos: Estos métodos son simples de entender y de programar ya que son iterativos, simples ciclos y sentencias que hacen que el vector pueda ser ordenado.
Recursivos: Estos métodos son aún más complejos, requieren de mayor atención y conocimiento para ser entendidos. Son rápidos y efectivos, utilizan generalmente la técnica Divide y vencerás, que consiste en

dividir un problema grande en varios pequeños para que sea más fácil resolverlos. Mediante llamadas recursivas a si mismos, es posible que el tiempo de ejecución y de ordenación sea más óptimo.

Complejidad en los Métodos de ordenamiento

La complejidad de un algoritmo es la cantidad de trabajo realizado y se mide por el número de operaciones básicas que se han llevado a cabo.

Por lo cual podemos comenzar comparando algunos de los algoritmos que desarrollaremos en esta unidad:

INSERCIÓN	cuadrático
SHELLSORT	sub- cuadrático
QUICKSORT	logarítmico
MERGESORT	logarítmico

Ordenamiento por Burbuja (Bubble Sort) $O(n^2)$

La Ordenación de burbuja (Bubble Sort) es un sencillo algoritmo de ordenamiento. Funciona revisando cada elemento de la lista o arreglo que va a ser ordenado, intercambiándolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada.

Este algoritmo obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas "burbujas". También es conocido como el método del intercambio directo. Dado que solo usa comparaciones para operar elementos, se lo considera un algoritmo de comparación, siendo el más sencillo de implementar.

Algoritmo BURBUJA_1

```
Inicio
        desde (I=1 hasta (N-1)) hacer
                desde (J=1 hasta (N-1)) hacer
                        Si (X(j) > X(j+1)) entonces
                                AUX=X(j)
                                X(j) =X(j+1)
                                X(j+1)=AUX
                        fin_si
                fin_desde
        fin_desde
fin
```

Algoritmo BURBUJA_2

```
inicio
        desde (I=1 hasta (N-1)) hacer
                desde (J=1 hasta (N-I)) hacer
                        Si (X(j) > X(j+1)) entonces
                                AUX=X(j)
                                X(j) =X(j+1)
                                X(j+1)=AUX
                        fin_si
                fin_desde
        fin_desde
fin
```

Algoritmo BURBUJA_3

```
inicio
        MARCA=FALSO
        I=1
        mientras ((MARCA=FALSO) y (I< N-1)) hacer
                MARCA=VERDADERO
                desde (J=1 hasta (N-1)) hacer
                        Si (X(j) > X(j+1)) entonces
                                AUX=X(j)
                                X(j)=X(j+1)
                                X(j+1)=AUX
                                MARCA=FALSO
                        fin_si
                fin_desde
                I=I+1
        fin_mientras
fin
```

Ordenamiento por insersión (Insertion Sort) $\boldsymbol{O\left(n^2\right)\Omega(n)}$

En este método, también llamado **insertion sort**, cuando hay n elementos se supone que se tiene un segmento inicial del array ordenado, el paso general es aumentar la longitud del segmento ordenado insertando el elemento siguiente X en el lugar adecuado, esto se hace moviendo cada elemento del segmento ordenado a la derecha, hasta que se encuentra un elemento < X.

```
Algoritmo INSERCION
        desde (N hasta I=2) hacer
                X=L(I)
                J=I- 1
```

```
            Mientras ( (L(J) > X) and (J > 0)) hacer
                  L(J+1)=L(J)
                  J=J-1
            fin_mientras
            L(J+1)=X
      fin_desde
fin
```

Nota: Partimos del supuesto de que ya se agrando el Arreglo y el número insertado ya se encuentra en la última posición.

```
Algoritmo INSERCION-2
      inserto=0; I=N-1;
      mientras  (I>1) o (inserto=0) hacer
            si X>=L(I) entonces
                        L(I+1)=X;
                        inserto=1;
                  sino
                        L(I+1)=L(I);                        finsi
            I--;
            si (I=1) y (inserto=0) entonces
                  L(I)=X;
            finsi
      fin_mientras
Fin
```

Para hallar la complejidad del algoritmo, analizamos el peor, el mejor y el caso medio.
El peor caso, es decir, el mayor número de comparaciones, se daría cuando el bucle interno tiene que hacer el mayor número de veces (I-1) siendo I el valor del núcleo externo, es decir, cuando cada elemento examinado es menor que el resto y esto ocurrirá cuando el array esté ordenado en orden contrario al que deseamos.
En el caso medio las comparaciones que se harían serían: para cada elemento I examinando como mínimo realizo una comparación en el caso que sea mayor que el resto y como máximo N-I.

Ordenamiento Shell (Shell Sort) $O(n^2)$

El método se denomina Shell en honor de su inventor Donald Shell. Su implementación original, requiere $O(n^2)$ comparaciones e intercambios en el peor caso. Un cambio menor presentado en el libro de V. Pratt produce una implementación con un rendimiento de $O(n \log 2\ n)$ en el peor caso. Esto es mejor que las $O(n^2)$ comparaciones requeridas por algoritmos simples pero peor que el óptimo $O(n\ log\ n)$. Aunque es fácil desarrollar un sentido intuitivo de cómo funciona este algoritmo, es muy difícil analizar su tiempo de ejecución.

El Shell Sort es una generalización del ordenamiento por inserción, teniendo en cuenta dos observaciones:

-El ordenamiento por inserción es eficiente si la entrada está "casi ordenada".

-El ordenamiento por inserción es ineficiente, en general, porque mueve los valores sólo una posición cada vez.

El algoritmo **Shell Sort** mejora el ordenamiento por inserción comparando elementos separados por un espacio de varias posiciones. Esto permite que un elemento haga "pasos más grandes" hacia su posición esperada. Los pasos múltiples sobre los datos se hacen con tamaños de espacio cada vez más pequeños. El último paso del **Shell Sort** es un simple *ordenamiento por inserción*, pero para entonces, ya está garantizado que los datos del vector están casi ordenados.

La **secuencia de espacios** es una parte integral del ***Shell sort***. Cualquier secuencia incremental funcionaría siempre que el último elemento sea 1. El algoritmo comienza realizando un ordenamiento por inserción con espacio, siendo el espacio el primer número en la secuencia de espacios. Continúa para realizar un ordenamiento por inserción con espacio para cada número en la secuencia, hasta que termina con un espacio de 1. Cuando el espacio es 1, el ordenamiento por inserción con espacio es simplemente un ordenamiento por inserción ordinario, garantizando que la lista final estará ordenada.

Pasos a seguir

*Dividir la lista original en **n/2** grupos de dos, considerando un incremento o salto entre los elementos de **n/2**.

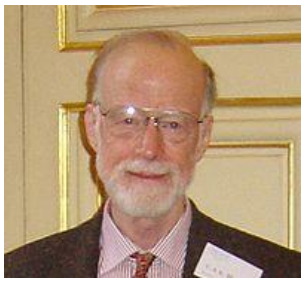
*Analizar cada grupo por separado, comparando las parejas de los elementos, y si no están ordenados, se intercambian.

*Se divide ahora la lista en la mitad de grupos **(n/4)**, con un incremento o salto entre los elementos también mitad **(n/4)**, y nuevamente se clasifica cada grupo por separado.

Así sucesivamente se sigue dividiendo la lista en la mitad de grupos que el recorrido anterior con un incremento de salto decreciente en la mitad que el salto anterior, y luego clasificando cada grupo por separado

El algoritmo termina cuando se consigue el tamaño de salto 1.

Ordenamiento Rápido (Quick Sort) $O(n \log n)$



El ordenamiento rápido *(Quicksort)*, o también ordenamiento por Partición, es un algoritmo creado por el científico británico en computación C. A. R. Hoare basado en la técnica de divide y vencerás, que permite, en promedio, ordenar **n** elementos en un tiempo proporcional a **n log n**.

El algoritmo: Se elige un elemento de la lista de elementos a ordenar, al que llamaremos **pivote**.
Se reacomodan los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el **pivote** *ocupa exactamente el lugar que le corresponderá en la lista ordenada*.
La lista queda **separada en dos sublistas**, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.
Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento.
Una vez terminado este proceso todos los elementos estarán ordenados.

La eficiencia del algoritmo depende de la posición en la que termine el pivote elegido.

- ***En el mejor caso***, el pivote termina en el centro de la lista, dividiéndola en dos sublistas de igual tamaño. En este caso, el orden de complejidad del algoritmo es O(n·log n).
- ***En el peor caso***, el pivote termina en un extremo de la lista. El orden de complejidad del algoritmo es entonces de $O(n^2)$. El peor caso dependerá de la implementación del algoritmo, aunque habitualmente ocurre en listas que se encuentran ordenadas, o casi ordenadas. Pero principalmente depende del pivote, si por ejemplo el algoritmo implementado toma como pivote siempre el primer elemento del array, y el array que le pasamos está ordenado, siempre va a generar a su izquierda un array vacío, lo que es ineficiente.
- ***En el caso promedio***, el orden es O(n·log n).

Técnicas de elección del pivote: El algoritmo básico del método Quicksort consiste en tomar cualquier elemento de la lista al cual denominaremos como pivote, dependiendo de la partición en que se elija, el algoritmo será más o menos eficiente.

- Tomar un elemento cualquiera como pivote tiene la ventaja de no requerir ningún cálculo adicional, lo cual lo hace bastante rápido. Sin embargo, esta elección *"a ciegas"* siempre provoca que el algoritmo tenga un orden de $O(n^2)$ para ciertas permutaciones de los elementos en la lista.
- Otra opción puede ser recorrer la lista para saber de antemano qué elemento ocupará la posición central de la lista, para elegirlo como pivote. Esto puede hacerse en **O(n)** y asegura que hasta en el peor de los casos, el algoritmo sea **O(n·log n)**. No obstante, el cálculo adicional rebaja bastante la eficiencia del algoritmo en el caso promedio.
- La opción a medio camino es tomar tres elementos de la lista - por ejemplo, el primero, el segundo, y el último - y compararlos, eligiendo el valor del medio como pivote.

```
Procedimiento QUICKSORT(Li,Ls:entero;L:array[1..N])
var PIVOTE:entero
inicio
        si (Li < Ls) entonces
                SUBLISTAS (Li,Ls,PIVOTE,L:array)
                QUICKSORT (Li,PIVOTE-1,L:array)
                QUICKSORT (PIVOTE+1,Ls,L:array)
                fin_si
        fin_quicksort
```

```
Procedimiento SUBLISTAS(FIRST,LAST,PIV:entero,L:array)
var X,I:entero
inicio
        X ☐ L(FIRST)
        PIV ☐FIRST
        desde (I=FIRST+1 hasta LAST)
                si (L(I) < X) entonces
                        (PIV ☐ PIV+1)
                        AUX ☐ L(PIV)
                        L(PIV)☐ L(I)
                        L(I) ☐ AUX
                fin_si
        fin_desde
        L(FIRST)☐ L(PIV)
        L(PIV) ☐ X
fin
```

```
Algoritmo PRIN_QUICKSORT
var a:array [1..N]
inicio
        leer A
        QUICKSORT(1,N,A
)
        escribir A
fin
```

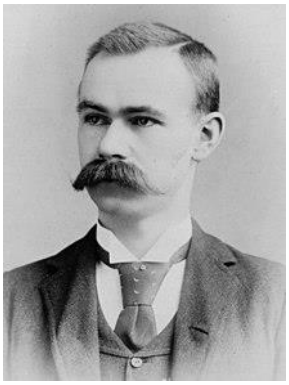
Algoritmo	**Operaciones máximas**
Burbuja	$O(N^2)$
Inserción	$O(N^2)/4$
Selección	$O(N^2)$
Shellsort	$O(N\ log^2 N)$
Mergesort	$O(N \log N)$
Quicksort	$O(N^2)$, en el peor y $N \log N$ en el promedio de casos

Ordenamiento Selección rápida

Una característica especial es la SELECCIÓN, la cual tiene una complejidad tal vez menor que el ordenamiento. Esto es debido principalmente a que debemos "encontrar" un elemento dentro de un arreglo y no tenemos la complejidad del ordenar al resto.
Lo ideal es poder utilizar el QUICKSORT como base y realizarle los ajustes necesarios.
Como vimos anteriormente, el QUICKSORT hace uso de dos llamadas recursivas, por lo cual en este caso sólo necesitaríamos una. Además podemos entender que en el peor de los casos de la SELECCIÓN RAPIDA estaríamos con las características del QUICKSORT.
Nuevamente, la elección del pivote, es el factor de éxito de este proceso.

Radix Sort

En ciencias de la computación, el **ordenamiento por base** *o* ***RADIX Sort****,* es un algoritmo de ordenación no comparativo. Evita la comparación creando y distribuyendo elementos en cubos de acuerdo con su base, que ordena enteros procesando sus dígitos de forma individual.
Para elementos con más de un dígito significativo, este proceso de agrupamiento se repite para cada dígito, mientras se conserva el orden del paso anterior, hasta que se hayan considerado todos los dígitos. Por esta razón, la ordenación por radix también se ha denominado ordenación por cubeta y ordenación digital.



Herman Hollerith fue el fundador de la compañía de máquinas de tabulación que se fusionó en 1911 con otras tres compañías para formar una quinta parte de la empresa, la Computing Tabulating Recording Company, más tarde llamada International Business Machines (IBM).

Como los enteros pueden representar cadenas de caracteres (por ejemplo, nombres o fechas) y, especialmente, números en punto flotan te especialmente formateados, radix sort no está limitado solo a los enteros. Es decir que este ordenamiento se puede aplicar a datos que se pueden ordenar lexicográficamente, ya sean números enteros, palabras, tarjetas perforadas, naipes o el correo.

Un clasificador de tarjetas de IBM que realiza un ordenamiento RADIX en un gran conjunto de tarjetas perforadas. Las tarjetas se introducen en una tolva debajo de la barbilla del operador y se clasifican en una de las 13 canastas de salida de la máquina, según los datos perforados en una columna de las tarjetas.

El **Radix Sort** se remonta a 1887 con el trabajo en máquinas de tabulación. Los algoritmos de ordenamiento por Radix se hicieron de uso común como una forma de clasificar tarjetas perforadas ya en 1923.
El primer algoritmo informático *(con uso eficiente de memoria)* fue desarrollado en 1954 en el MIT por Harold H. Seward. Los tipos de radix computarizados se habían descartado anteriormente como poco prácticos debido a la necesidad percibida de una asignación variable de cubos de tamaño desconocido. La innovación de Seward fue utilizar un escaneo lineal para determinar de antemano los tamaños de cubeta y las compensaciones requeridas, lo que permite una única asignación estática

de memoria auxiliar. El escaneo lineal está estrechamente relacionado con el otro algoritmo de Seward: el tipo de conteo.
En la era moderna, las clases de radix se aplican más comúnmente a colecciones de cadenas binarias y enteros. Se ha demostrado en algunos puntos de referencia que es más rápido que otros algoritmos de clasificación de propósito más general, a veces entre un 50% y tres veces más rápido.

La mayor parte de los métodos de ordenamiento representan internamente todos sus datos como números binarios, por lo que cambiar las representaciones de enteros por representaciones de grupos de dígitos binarios es lo más conveniente. Existen dos clasificaciones de radix sort: el de dígito menos significativo (LSD) y el de dígito más significativo (MSD). Radix sort LSD procesa las representaciones de enteros empezando por el dígito menos significativo y moviéndose hacia el dígito más significativo. Radix sort MSD trabaja en sentido contrario.

Radix sort LSD usa típicamente el siguiente orden: claves cortas aparecen antes que las claves largas, y claves de la misma longitud son ordenadas de forma léxica. Esto coincide con el orden normal de las representaciones de enteros, como la secuencia "1, 2, 3, 4, 5, 6, 7, 8, 9, 10".
Radix sorts MSD usa orden léxico, que es ideal para la ordenación de cadenas de caracteres, como las palabras o representaciones de enteros de longitud fija. Una secuencia como "b, c, d, e, f, g, h, i, j, ba" será ordenada léxicamente como "b, ba, c, d, e, f, g, h, i, j". Si se usa orden léxico para ordenar representaciones de enteros de longitud variable, entonces la ordenación de las representaciones de los números del 1 al 10 será "1, 10, 2, 3, 4, 5, 6, 7, 8, 9", como si las claves más cortas estuvieran justificadas a la izquierda y rellenadas a la derecha con espacios en blanco, para hacerlas tan largas como la clave más larga.

Aparte del orden transversal, los tipos MSD y LSD difieren en su manejo de la entrada de longitud variable. Los tipos de LSD pueden agrupar por longitud, ordenar por base a cada grupo y luego concatenar los grupos en orden de tamaño. Las clasificaciones de MSD deben "extender" todas las claves más cortas al tamaño de la clave más grande y ordenarlas en consecuencia, lo que puede ser más complicado que la agrupación requerida por LSD.
Sin embargo, los tipos de MSD son más susceptibles de subdivisión y recursión. Cada segmento creado por un paso de MSD puede ordenarse por sí mismo utilizando el siguiente dígito más significativo, sin referencia a ningún otro segmento creado en el paso anterior. Una vez que se alcanza el último dígito, concatenar los depósitos es todo lo que se requiere para completar la clasificación.

Ejemplo LSD:
Lista de entrada (base 10):
[170, 45, 75, 90, 2, 802, 2, 66]
Comenzando desde el (último) dígito más a la derecha, ordene los números según ese dígito:

[{170, 90}, {2, 802, 2}, {45, 75}, {66}]
Ordenar por el siguiente dígito de la izquierda:

[{02, 802, 02}, {45}, {66}, {170, 75}, {90}]
Observe que se antepone un dígito 0 implícito a los dos 2 para que 802 mantenga su posición entre ellos.
Y finalmente por el dígito más a la izquierda:

[{002, 002, 045, 066, 075, 090}, {170}, {802}]
Observe que se antepone un 0 a todos los números de 1 o 2 dígitos.
Cada paso requiere solo una pasada sobre los datos, ya que cada elemento se puede colocar en su contenedor sin compararlo con ningún otro elemento.

Algunas implementaciones del ordenamiento Radix asignan espacio para los depósitos contando primero la cantidad de claves que pertenecen a cada depósito antes de mover las claves a esos depósitos. El número de veces que ocurre cada dígito se almacena en una matriz.

Aunque siempre es posible predeterminar los límites del depósito mediante recuentos, algunas implementaciones optan por utilizar la asignación de memoria dinámica en su lugar.

Ejemplo MSD *(Recursivo hacia adelante)*:
Lista de entrada, cadenas numéricas de ancho fijo con ceros a la izquierda:

[170, 045, 075, 025, 002, 024, 802, 066]
Primer dígito, con paréntesis que indican cubos:

[{045, 075, 025, 002, 024, 066}, {170}, {802}]
Tenga en cuenta que 170 y 802 ya están completos porque son todo lo que queda en sus depósitos, por lo que no se necesita más recursividad
Siguiente dígito:

[{{002}, {025, 024}, {045}, {066}, {075}}, 170, 802]
Dígito final:

[002, {{024}, {025}}, 045, 066, 075, 170, 802]
Todo lo que queda es la concatenación:

[002, 024, 025, 045, 066, 075, 170, 802]

Complejidad:
La ordenación por radix opera en tiempo O (nw), donde n es el número de claves yw es la longitud de la clave. Las variantes de LSD pueden lograr un límite inferior para w de 'longitud promedio de clave' al dividir claves de longitud variable en grupos como se discutió anteriormente.
Las clasificaciones de radix optimizadas pueden ser muy rápidas cuando se trabaja en un dominio que les conviene. Están restringidos a datos lexicográficos, pero para muchas aplicaciones prácticas esto no es una limitación. Los tamaños de clave grandes pueden obstaculizar las implementaciones de LSD cuando el número inducido de pasadas se convierte en el cuello de botella.

Otros algoritmos

Los conceptos se utilizan para valorar las páginas, se dividen en dos grandes familias:

1.- Criterios internos

2.- Criterios externos

Tanto los conceptos internos a la propia página web como los externos son importantes, algunos de los conceptos incluso no se valoran directamente, si no que permiten que la página consiga una mejor valoración para otras.

Hay que saber qué es lo que se pretende con una página antes de definir qué estrategia o combinación de estrategias se aplicará.

Criterios internos a la página web:

A grandes rasgos, consisten en todo aquello que se encuentra en la propia página, entre ellos destacan:

1. Número de pantallas: (además pantallas más valoración)
2. Títulos de las pantallas: (las pantallas deben tener un título)
3. Redacción de los textos
4. Estructura que permita una fácil navegación
5. Tecnologías aplicadas: (evitar utilización abusiva de flash, javascripts...)
6. Direcciones de las pantallas de acuerdo con el contenido: (url descriptivas).

Criterios externos a la página web: Son los conceptos que los buscadores interpretan según cuanto más enlaces tiene una página, más relevancia merece, y por lo tanto aparecerá más arriba en las búsquedas relacionadas.

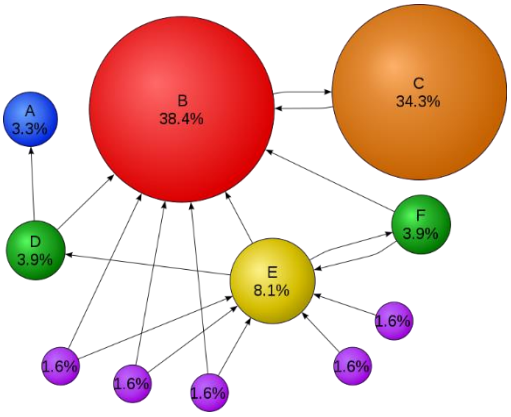
Estos enlaces pueden realizarse de varias maneras, aun cuando algunos no tienen valor para los buscadores, la explicación en este caso se limitará a aquellas que si que tienen. De un lado, existe la posibilidad de enlaces mediante imágenes (.GIF o bien .JPG); estas imágenes, si no contienen textos alternativos ni descripción, simplemente votan a la página de destino, incrementando el valor de cara a los motores de búsqueda.

El otro tipo de enlace, es el enlace de texto, que al cliquear encima, lleva hacia la página web de destino. Este tipo de enlace es interesante porque potencia la página de destino por el texto exacto del enlace, permitiendo potenciar exclusivamente aquel texto, además de también incrementar el valor de la página de destino por su propio contenido.

Por este motivo, si hay muchas páginas que enlazan una página, los buscadores interpretan que cada página web emite su voto para aquella a la que va destinado el enlace, dándole más y más preferencia.

De este modo, las páginas destinatarias de enlaces, ganan posiciones por ser consideradas como más importantes por aquellos términos por los que están referidas, aún así, es necesario saber cómo deben ser dichos enlaces para optimizar su utilidad.

PageRank es una marca registrada y patentada por *Google el 9 de enero de 1999* que ampara una familia de algoritmos utilizados para asignar de forma numérica la relevancia de los documentos (o páginas web) indexados por un motor de búsqueda. Sus propiedades son muy discutidas por los expertos en optimización de motores de búsqueda. El sistema PageRank es utilizado por el popular motor de búsqueda Google para ayudarle a determinar la importancia o relevancia de una página. Fue desarrollado por los fundadores de Google, Larry Page *(apellido, del cual, recibe el nombre este algoritmo)* y Sergey Brin, en la Universidad de Stanford mientras estudiaban el posgrado en ciencias de la computación.



PageRank confía en la naturaleza democrática de la web utilizando su vasta estructura de enlaces como un indicador del valor de una página en concreto. Google interpreta un enlace de una página A a una página B como un voto, de la página A, para la página B. Pero Google mira más allá del volumen de votos, o enlaces que una página recibe; también analiza la página que emite el voto. Los votos emitidos por las páginas consideradas "importantes", es decir con un PageRank elevado, valen más, y ayudan a hacer a otras páginas "importantes". Por lo tanto, el PageRank de una página refleja la importancia de la misma en Internet.

$$\mathrm{PR}(A) = (1-d) + d\sum_{i=1}^{n}\frac{\mathrm{PR}(i)}{C(i)}$$

Donde:

- $\mathrm{PR}(A)$ es el PageRank de la página A.
- **d** es un factor de amortiguación que tiene un valor entre 0 y 1.
- $\mathrm{PR}(i)$ son los valores de PageRank que tienen cada una de las páginas **i** que enlazan a A.
- **C(i)** es el número total de enlaces salientes de la página i (sean o no hacia A).